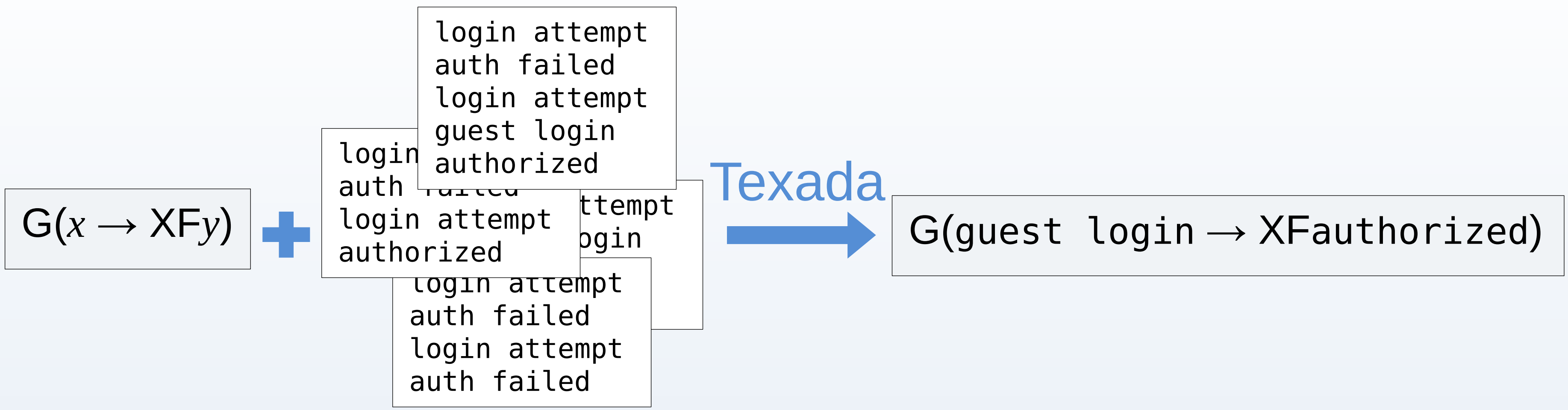


General LTL Specification Mining

Caroline Lemieux, Dennis Park and Ivan Beschastnikh

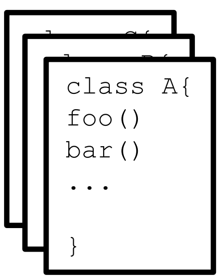
University of British Columbia
Department of Computer Science

source: **https://bitbucket.org/bestchai/texada**

Program Specifications

- Formal expectation of how a program should work
- Specs are useful, but **rarely specified by developers**
  - May be difficult to write out
  - May fall out of date like documentation

program without specs:
easier for initial dev



harder for debugging, refactoring, maintenance

program with specs:
harder for initial dev

easier for debugging, refactoring, maintenance

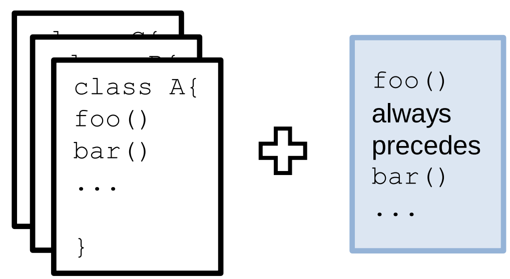
Program Specifications

- Formal expectation of how a program should work
- Specs are useful, but **rarely specified by developers**
  - May be difficult to write out
  - May fall out of date like documentation

program without specs:
easier for initial dev

harder for debugging, refactoring, maintenance

program with specs:
harder for initial dev

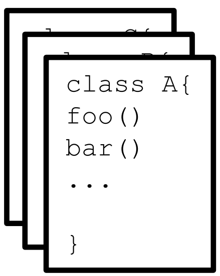


easier for debugging, refactoring, maintenance

Program Specifications

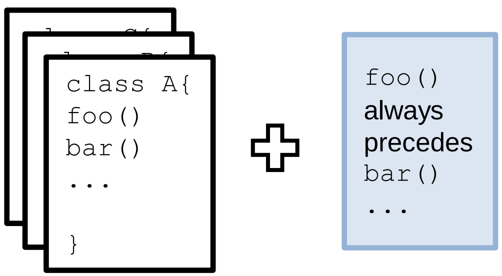
- Formal expectation of how a program should work
- Specs are useful, but **rarely specified by developers**
  - May be difficult to write out
  - May fall out of date like documentation

program without specs:
easier for initial dev



harder for debugging, refactoring, maintenance

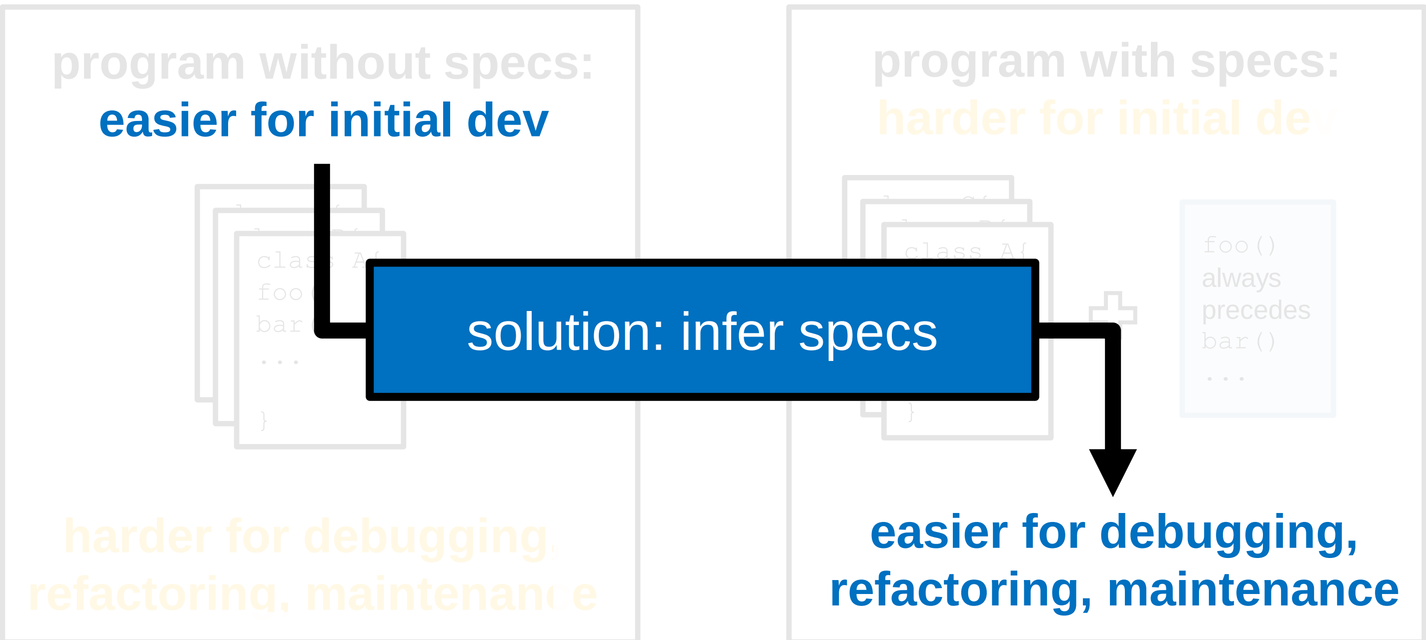
program with specs:
harder for initial dev



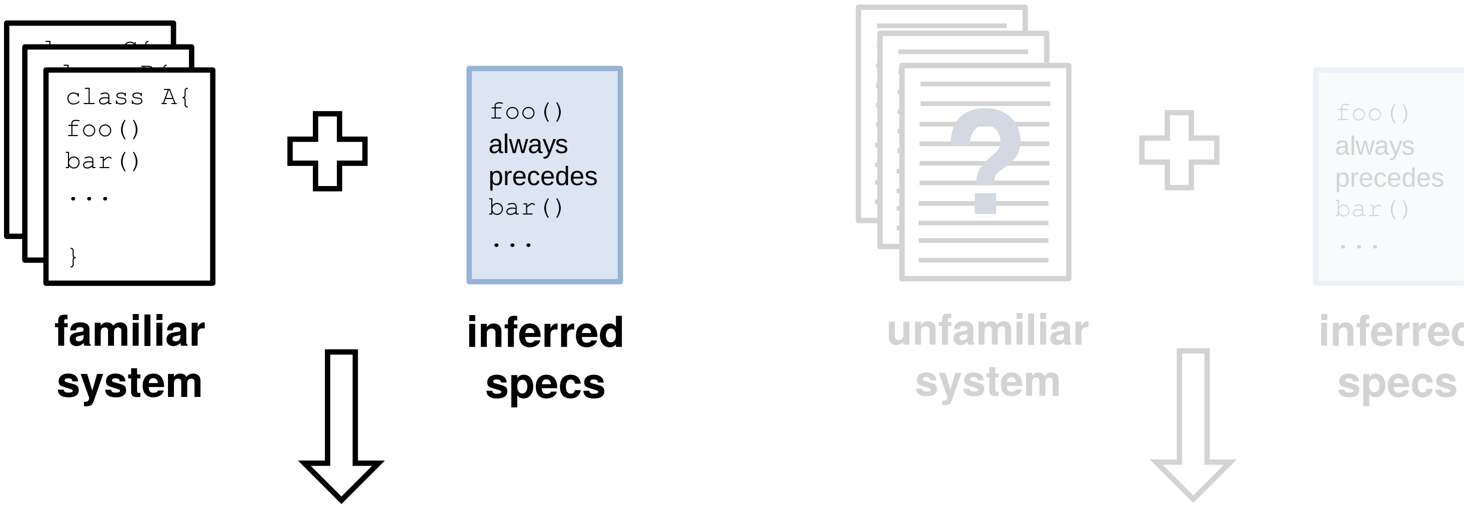
easier for debugging, refactoring, maintenance

Program Specifications

- Formal expectation of how a program should work
- Specs are useful, but **rarely specified by developers**
  - May be difficult to write out
  - May fall out of date like documentation



Uses of Inferred Specs in Familiar Systems

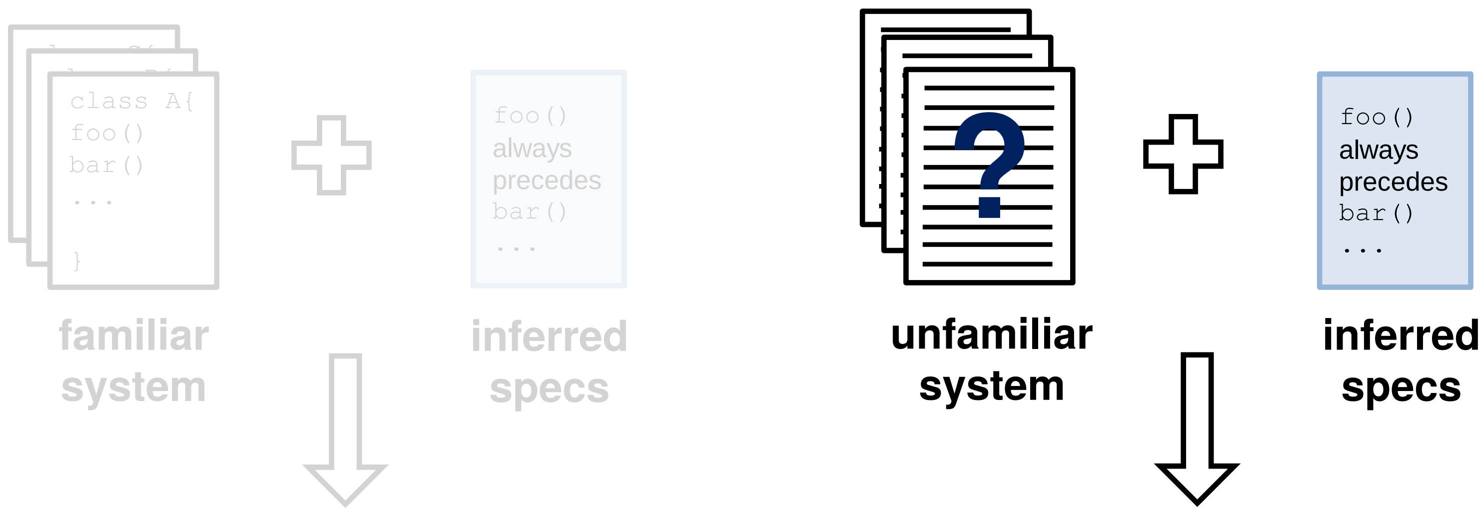


- program maintenance[1]
- confirm expected behavior[2]
- bug detection[2]
- test generation[3]

- system comprehension[4]
- system modeling[4]
- reverse engineering[1]

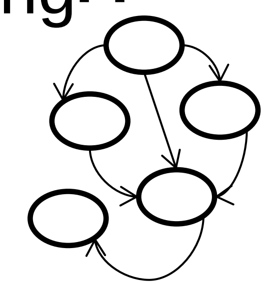
[1] M. P. Robillard, E. Bodden, D. Kawrykow, M. Mezini, and T. Ratchford. Automated API Property Inference Techniques. TSE, 613-637, 2013.
[2] M. D. Ernst, J. Cockrell, W. G. Griswold and D. Notkin. Dynamically Discovering Likely Program Invariants to Support program evolution. TSE, 27(2):99–123, 2001.
[3] V Dallmeier, N. Knopp, C. Mallon, S. Hack and A. Zeller. Generating Test Cases for Specification Mining. ISSTA, 85-96, 2010.
[4] I. Beschastnikh, Y. Brun, S. Schneider, M. Sloan and M. D. Ernst .Leveraging existing instrumentation to automatically infer invariant-constrained models. FSE, 267–277, 2011.

Inferred Specs in Unfamiliar Systems



- program maintenance[1]
- confirm expected behavior[2]
- bug detection[2]
- test generation[3]

- system comprehension[4]
- system modeling[4]
- reverse engineering[1]



[1] M. P. Robillard, E. Bodden, D. Kawrykow, M. Mezini, and T. Ratchford. Automated API Property Inference Techniques. TSE, 613-637, 2013.
[2] M. D. Ernst, J. Cockrell, W. G. Griswold and D. Notkin. Dynamically Discovering Likely Program Invariants to Support program evolution. TSE, 27(2):99–123, 2001.
[3] V Dallmeier, N. Knopp, C. Mallon, S. Hack and A. Zeller. Generating Test Cases for Specification Mining. ISSTA, 85-96, 2010.
[4] I. Beschastnikh, Y. Brun, S. Schneider, M. Sloan and M. D. Ernst .Leveraging existing instrumentation to automatically infer invariant-constrained models. FSE, 267–277, 2011.

Spec Mining Sources

- Specs can be mined from various program artifacts.
  - Source code [1]
  - Documentation [2]
  - Revision histories [3]
- Focus of talk: textual logs (e.g., execution traces)
  - Easy to instrument, extensible

web log

```
sales_page
search
sales_anncs
search
sales_anncs
search
search
sales_anncs
sales_anncs
..
homepage
search
homepage
search
sales_anncs
sales_anncs
homepage
search
```

dining phil.

```
0 is THINKING
1 is HUNGRY
2 is THINKING
3 is THINKING
4 is THINKING
..
0 is THINKING
1 is HUNGRY
2 is THINKING
3 is THINKING
4 is THINKING
..
0 is THINKING
1 is THINKING
2 is THINKING
3 is THINKING
4 is THINKING
..
```

data struct.

```
StackAr(int)
isFull()
isEmpty()
top()
isEmpty()
topAndPop()
isEmpty()
isFull()
isEmpty()
top()
isEmpty()
push(java.lang.Object)
isFull()
isFull()
isEmpty()
top()
isEmpty()
push(java.lang.Object)
```

data inv. log

```
this.currentSize == this.front
this.currentSize == this.back
this.theArray[] elements == null
this.theArray[].getClass() elements == null
this.currentSize == 0
..
this.back <= size(this.theArray[])-1
..
this.back <= size(this.theArray[])-1
..
this.back <= size(this.theArray[])-1
..
this.back <= size(this.theArray[])-1
..
this.theArray[] elements == null
this.theArray[].getClass() elements == null
this.currentSize == 0
this.front one of { 0, 6 }
```


[1] R. Alur, P. Cerny, P. Madhusudan, W. Nam. Synthesis of Interface Specifications for Java Classes. In Proceedings of POPL'05.
[2]L. Tan, D. Yuan, G. Krishna, and Y. Zhou. /*Icomment: Bugs or BadComments?*/. In Proceedings of SOSP'07.
[3] V. B. Livshits and T. Zimmermann. Dynamine: Finding Common Error Patterns by Mining Software Revision Histories. In Proceedings of ESEC/FSE'05.

Spec Patterns to Mine

- In this talk, focus on mining temporal specs
  - `open()` is always followed by `close()` (response pattern)
- Many temporal properties could be mined:

strict response pattern + resource allocation [2]

lots of small patterns to combine into big ones [4]

…

variations of response pattern [1]

response patterns of arbitrary length [3]

branching live-sequence charts [5]

[1] J. Yang, D. Evans, D. Bhardwaj, T. Bhat and M. Das. Perracotta: Mining Temporal API Rules from Imperfect Traces. ICSE'06.
[2] M. Gabel and Z. Su. Javert: Fully Automatic Mining of General Temporal Properties from Dynamic Traces. FSE'08.
[3] D. Lo, S-C. Khoo, and C. Liu. Mining Temporal Rules for Software Maintenance. Journal of Software Maintenance and Evolution: Research and Practice, 20 (4), 2008.
[4] G. Reger, H. Barringer, and D. Rydeheard. A Pattern-Based Approach to Parametric Specification Mining. In Proceedings of ASE'13.
[5] D. Fahland, D. Lo, and S. Maoz. Mining Branching-Time Scenarios. In Proceedings of ASE'13.

Spec Patterns to Mine

- In this talk, focus on mining temporal specs
  - `open()` is always followed by `close()` (response pattern)
- Many temporal properties could be mined:

strict response

lots of small

Which temporal spec mining tool should I use?

variations of response pattern [1]

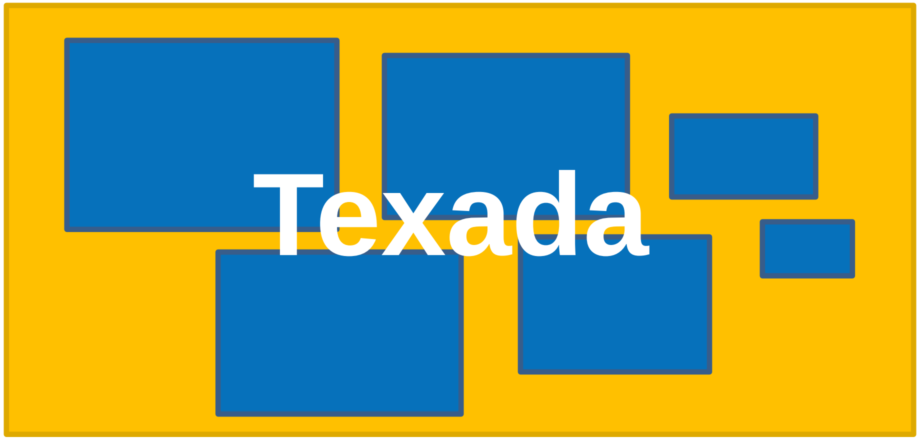
response patterns of arbitrary length [3]

branching live-sequence charts [5]

[1] J. Yang, D. Evans, D. Bhardwaj, T. Bhat and M. Das. Perracotta: Mining Temporal API Rules from Imperfect Traces. ICSE'06.
[2] M. Gabel and Z. Su. Javert: Fully Automatic Mining of General Temporal Properties from Dynamic Traces. FSE'08.
[3] D. Lo, S-C. Khoo, and C. Liu. Mining Temporal Rules for Software Maintenance. Journal of Software Maintenance and Evolution: Research and Practice, 20 (4), 2008.
[4] G. Reger, H. Barringer, and D. Rydeheard. A Pattern-Based Approach to Parametric Specification Mining. In Proceedings of ASE'13.
[5] D. Fahland, D. Lo, and S. Maoz. Mining Branching-Time Scenarios. In Proceedings of ASE'13.

“Ultimate” Temporal Spec Inference

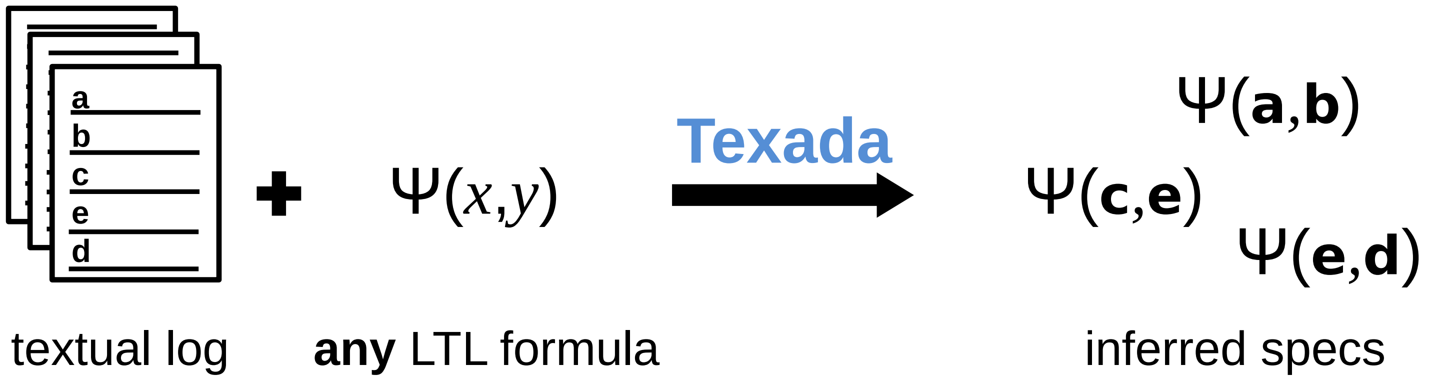
mine any general temporal pattern



- **pattern-based:** can output a set of simple patterns, or more general patterns
- **patterns specified in LTL**, includes 67 pre-defined templates

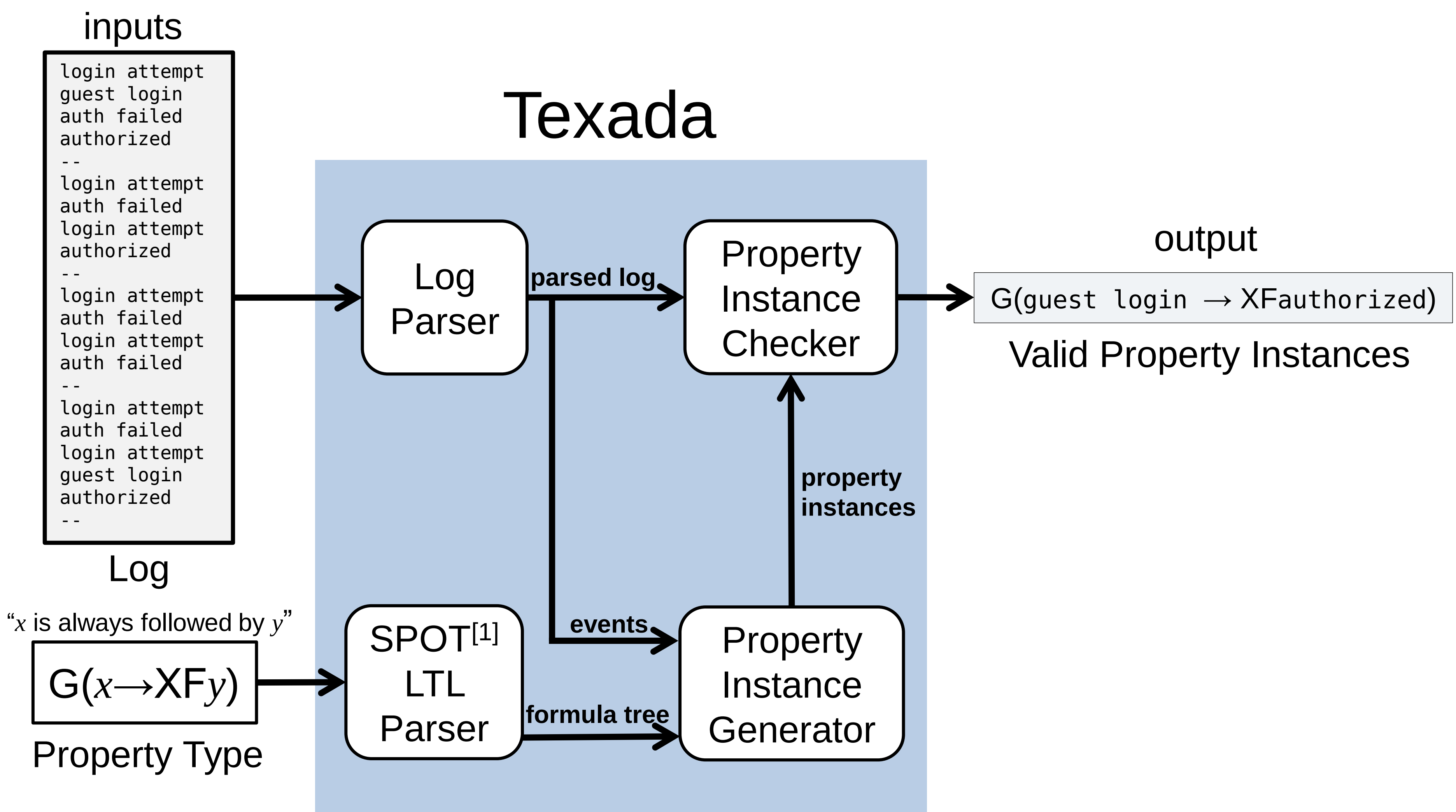
Contributions

- **Texada**: general LTL specification miner



- Approximate confidence/support measures for LTL
- Concurrent system analysis
  - Dining Philosophers
  - Sleeping Barber

Texada Outline



[1] A. Duret-Lutz and D. Poitrenaud. Spot: an Extensible Model Checking Library using Transition-Based Generalized Buchi automata. In Proceedings of MASCOTS'04.

Property Type Mining

- Parse each property type into interpretable format (tree)
- For each property type, dynamically generate and check property instances on log:

“*x* is always followed by *y*”

$$G(x \rightarrow XFy)$$

G(`authorized` → XF`guest login`)	✗
G(`authorized` → XF`login attempt`)	✗
G(`authorized` → XF`auth failed`)	✗
G(`guest login` → XF`authorized`)	✓
G(`auth failed`→ XF`authorized`)	✗
G(`auth failed`→ XF`guest login`)	✗
G(`auth failed` → XF`authorized`)	✗
G(`guest login`→ XF`login attempt`)	✗
G(`guest login`→ XF`auth failed`)	✗
G(`login attempt` → XF`authorized`)	✗
G(`login attempt` → XF`guest login`)	✗
G(`login attempt` → XF`auth failed`)	✗

Linear Log Parsing

Texada parses logs by regexes (specify event line format, trace separator)

```
login attempt
guest login
auth failed
authorized
--
login attempt
auth failed
login attempt
authorized
--
login attempt
auth failed
login attempt
auth failed
--
login attempt
auth failed
login attempt
guest login
authorized
--
```

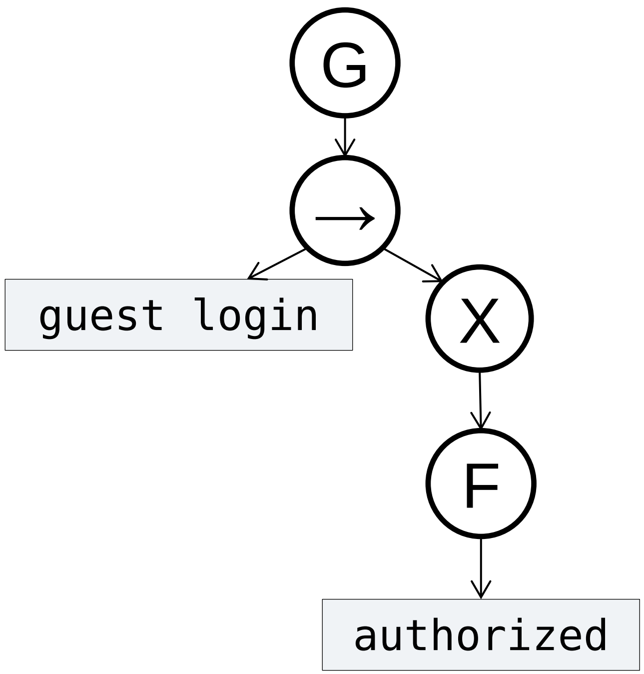
set of traces in linear format

1.	login attempt	guest login	auth failed	authorized	
2.	login attempt	auth failed	login attempt	authorized	
3.	login attempt	auth failed	login attempt	auth failed	
4.	login attempt	auth failed	login attempt	guest login	authorized

Property Instance Checking (Linear Alg)

- Check each instance on each trace in log
- holds on trace ⇔ holds on first event of trace

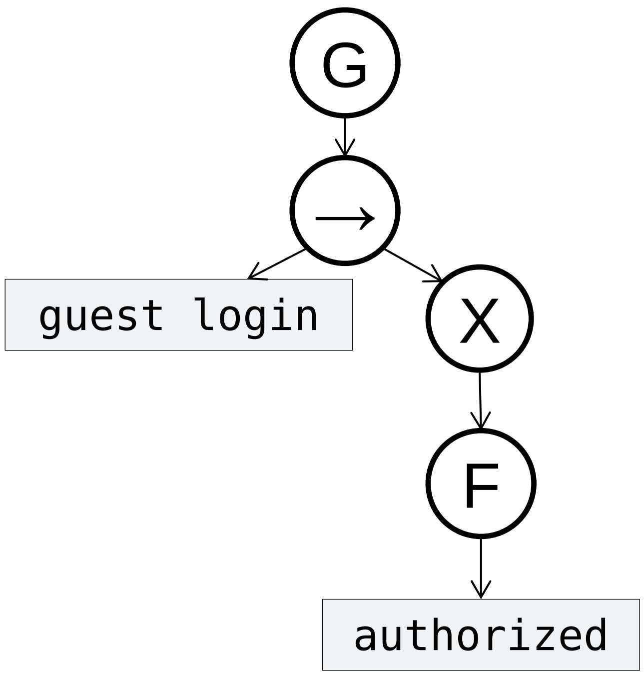
0	1	2	3
guest login	login attempt	auth failed	authorized



Property Instance Checking (Linear Alg)

- Check each instance on each trace in log
- holds on trace ⇔ holds on first event of trace

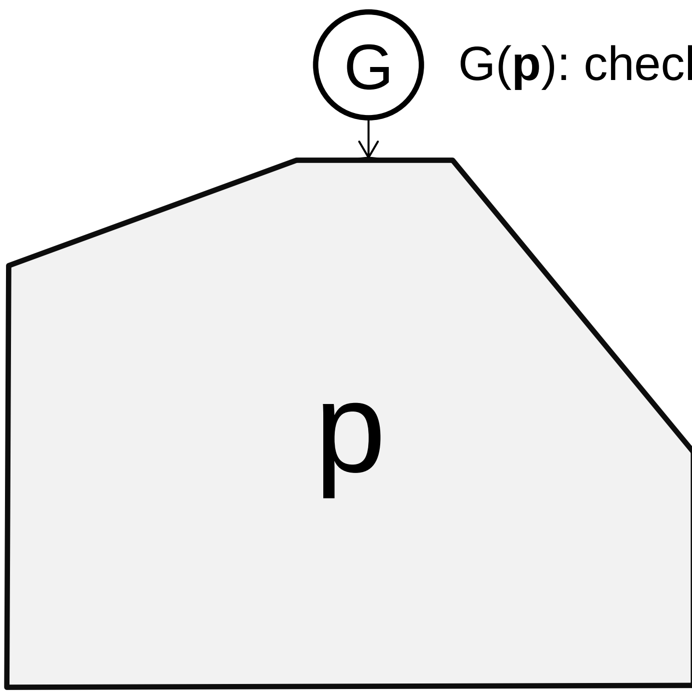
0	1	2	3
guest login	login attempt	auth failed	authorized



Property Instance Checking (Linear Alg)

- Check each instance on each trace in log
- holds on trace ⇔ holds on first event of trace

0	1	2	3
guest login	login attempt	auth failed	authorized

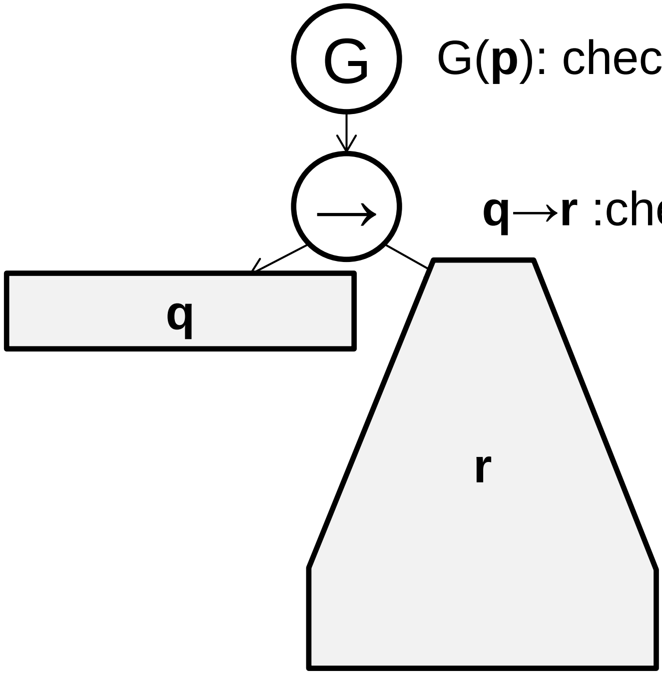


G(**p**): check if **p** holds at every time point

Property Instance Checking (Linear Alg)

- Check each instance on each trace in log
- holds on trace ⇔ holds on first event of trace

0	1	2	3
`guest login`	`login attempt`	`auth failed`	`authorized`

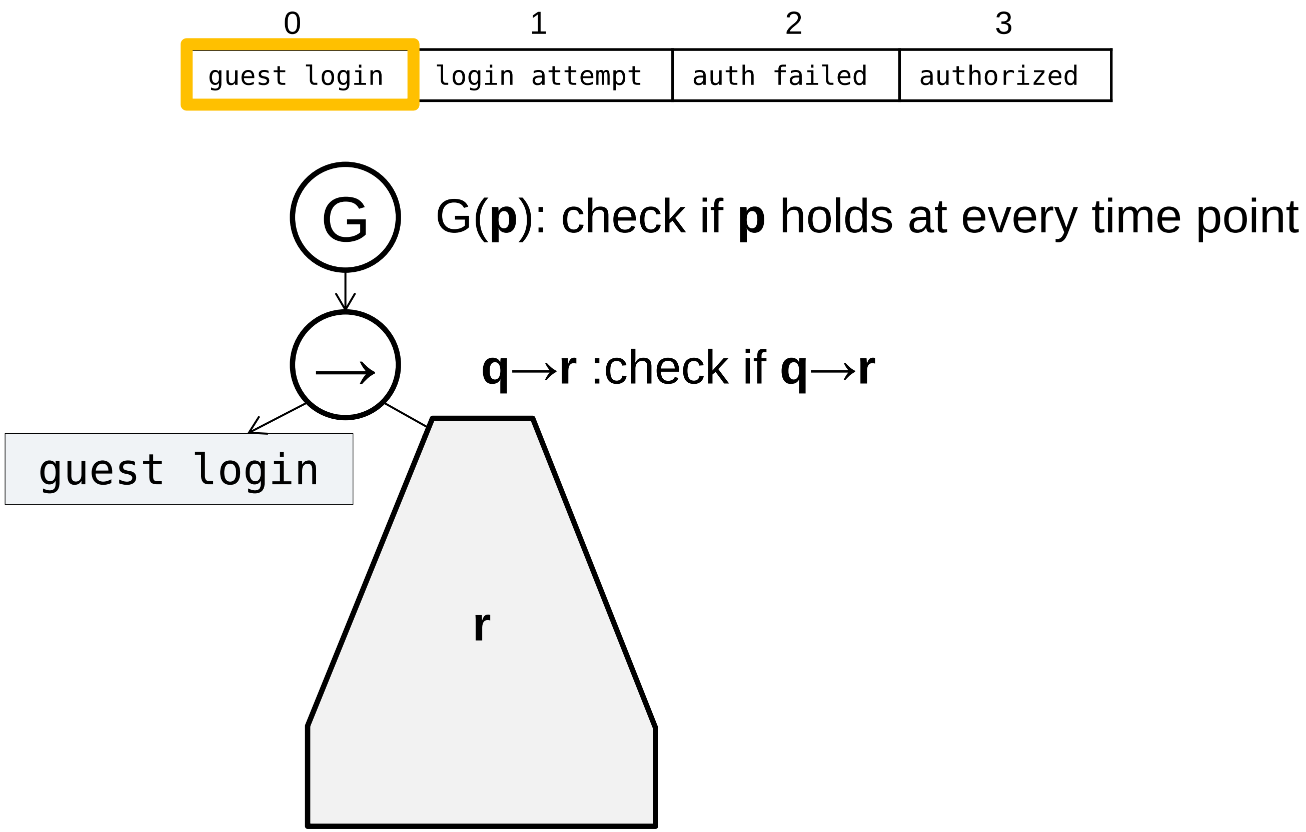


G(**p**): check if **p** holds at every time point

q→**r** :check if **q**→**r**

Property Instance Checking (Linear Alg)

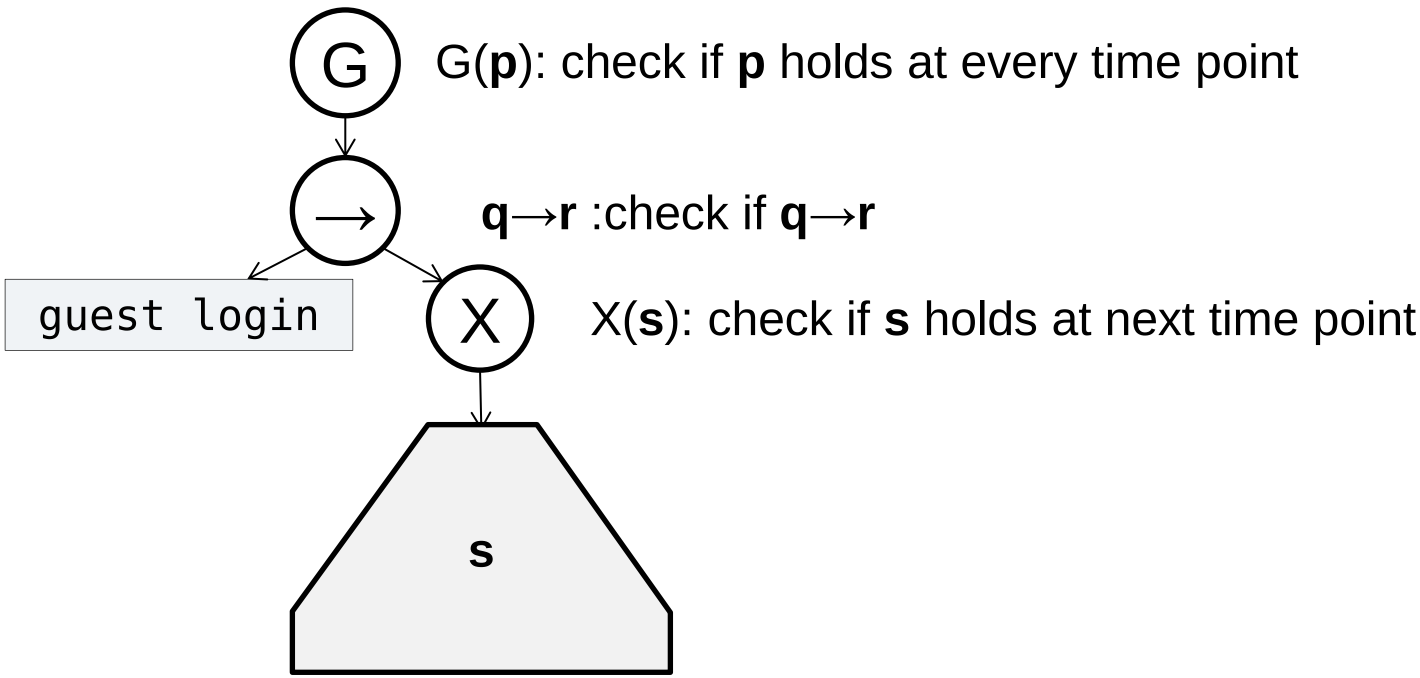
- Check each instance on each trace in log
- holds on trace ⇔ holds on first event of trace



Property Instance Checking (Linear Alg)

- Check each instance on each trace in log
- holds on trace ⇔ holds on first event of trace

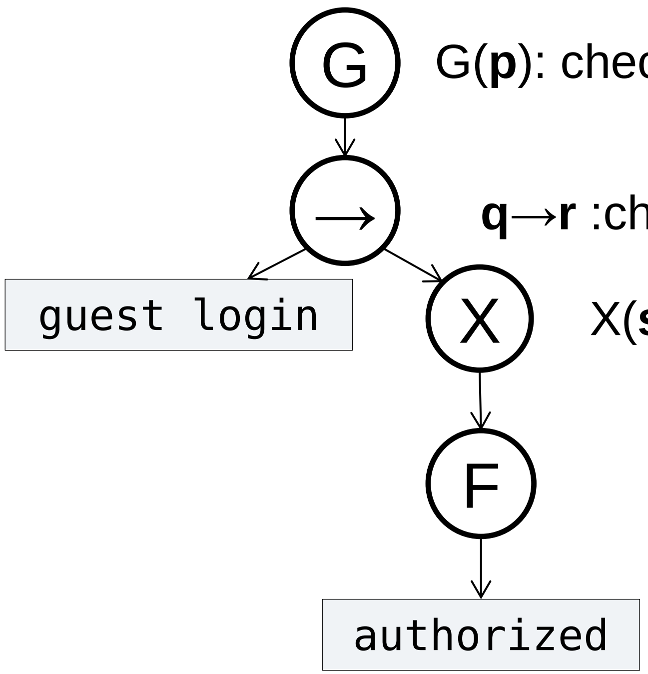
0	1	2	3
guest login	login attempt	auth failed	authorized



Property Instance Checking (Linear Alg)

- Check each instance on each trace in log
- holds on trace ⇔ holds on first event of trace

0	1	2	3
`guest login`	`login attempt`	`auth failed`	`authorized`



G(**p**): check if **p** holds at every time point

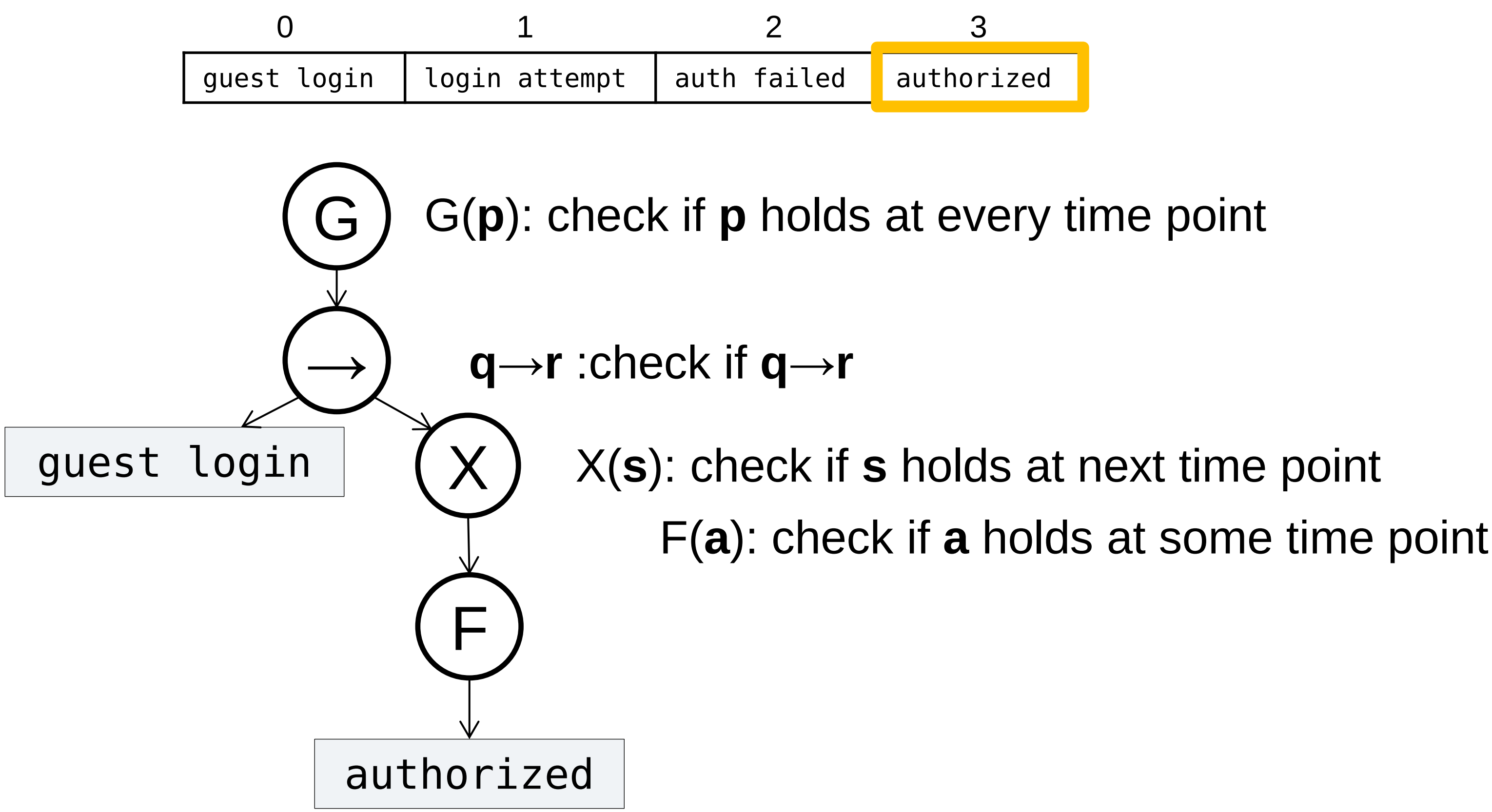
q→**r** :check if **q**→**r**

X(**s**): check if **s** holds at next time point

F(**a**): check if **a** holds at some time point

Property Instance Checking (Linear Alg)

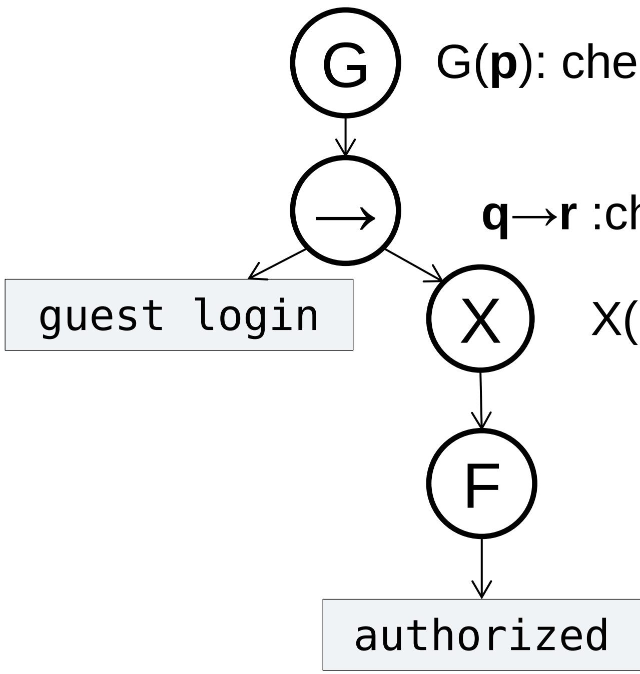
- Check each instance on each trace in log
- holds on trace ⇔ holds on first event of trace



Property Instance Checking (Linear Alg)

- Check each instance on each trace in log
- holds on trace ⇔ holds on first event of trace

0	1	2	3
guest login	login attempt	auth failed	authorized



G(**p**): check if **p** holds at every time point

q→**r** :check if **q**→**r**

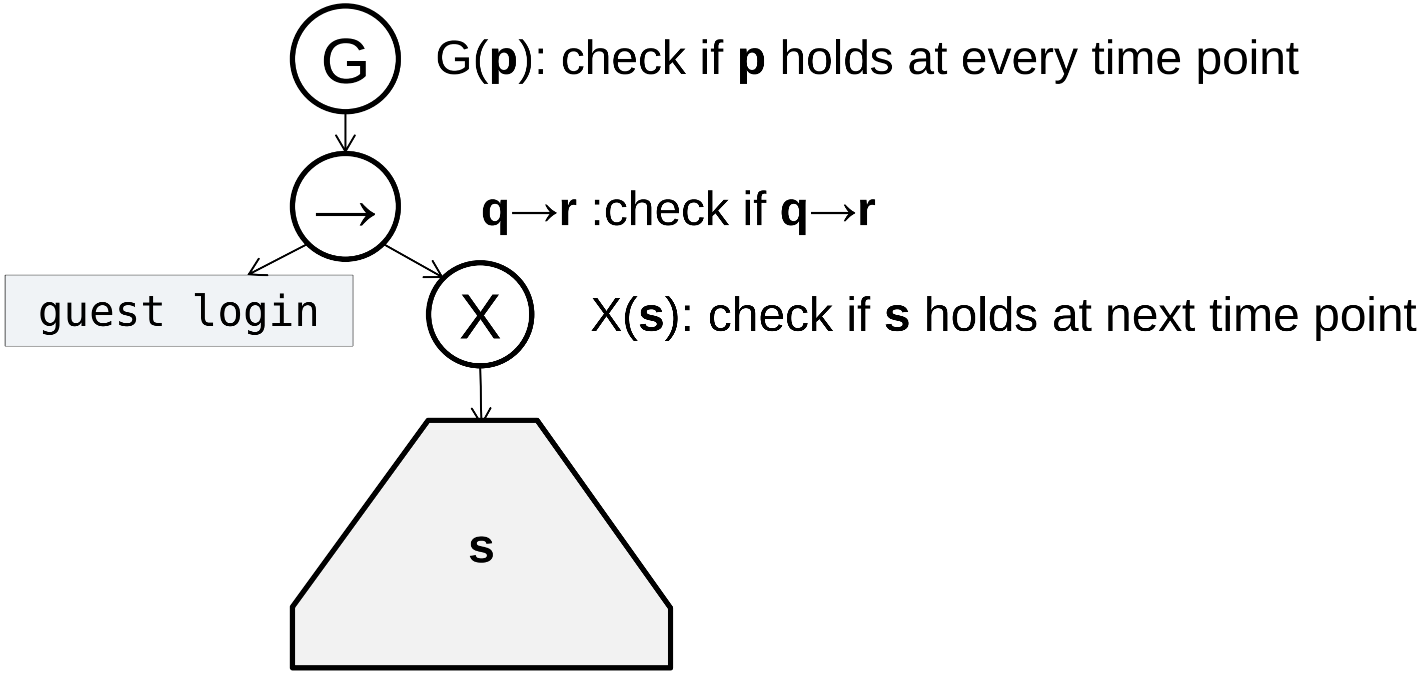
X(**s**): check if **s** holds at next time point

F(**a**): check if **a** holds at some time point

Property Instance Checking (Linear Alg)

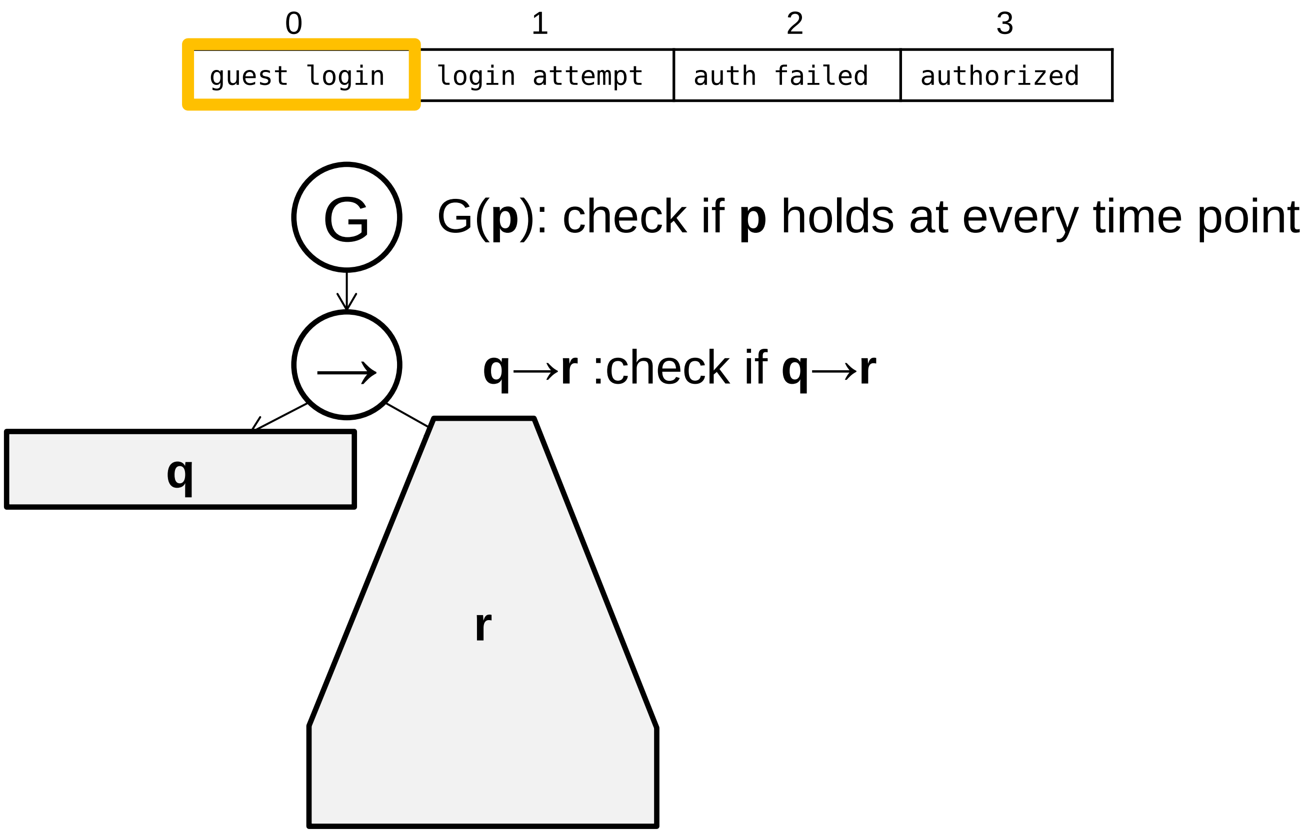
- Check each instance on each trace in log
- holds on trace ⇔ holds on first event of trace

0	1	2	3
guest login	login attempt	auth failed	authorized



Property Instance Checking (Linear Alg)

- Check each instance on each trace in log
- holds on trace ⇔ holds on first event of trace

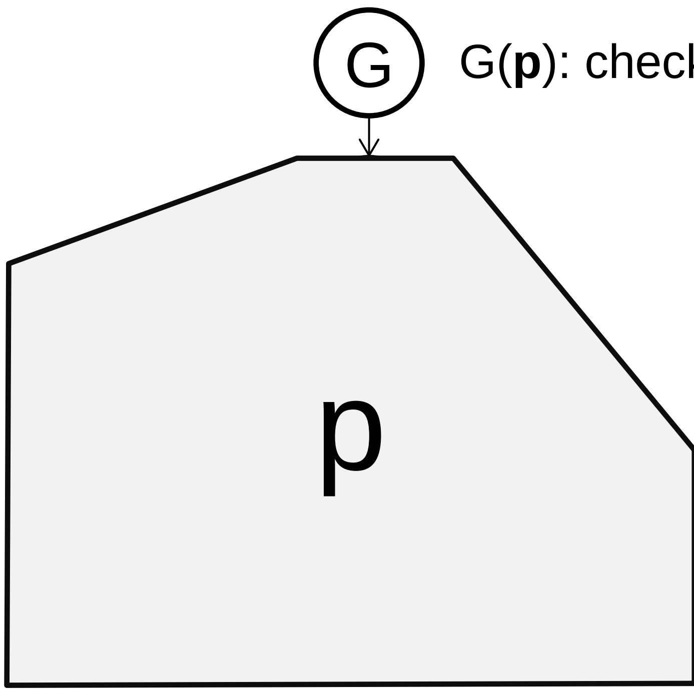


Property Instance Checking (Linear Alg)

- Check each instance on each trace in log
- holds on trace ⇔ holds on first event of trace

0	1	2	3
guest login	login attempt	auth failed	authorized

G(**p**): check if **p** holds at every time point



Linear Algorithm Observations

- Linear checker works but … is slow.
- Notice: most temporal operators rely on relative positions
- Optimization: use **map format**

`login attempt`	`guest login`	`auth failed`	`authorized`

→

event	**posns**
`login attempt`	`[0]`
`guest login`	`[1]`
`auth failed`	`[2]`
`authorized`	`[3]`

`login attempt`	`auth failed`	`login attempt`	`auth failed`

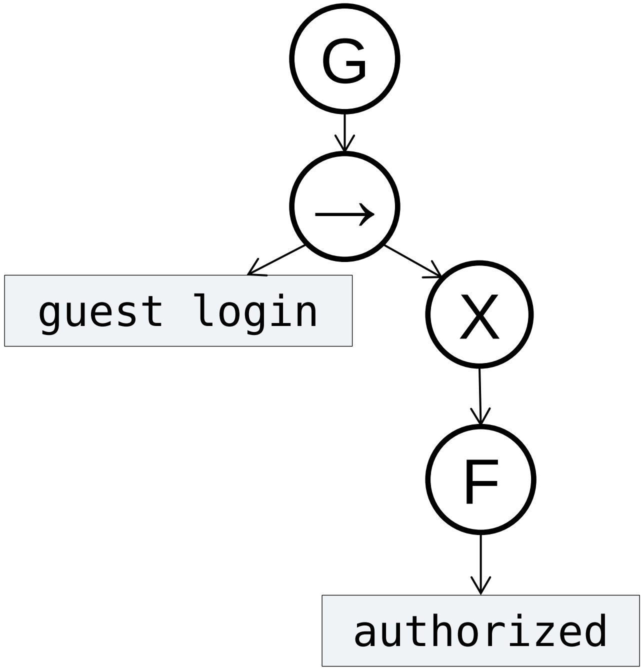
→

event	**posns**
`login attempt`	`[0,2]`
`auth failed`	`[1,3]`

Checking on Map Traces

- Check on trace in map form also tree-based
  - but also uses the negation of nodes
- Map form allows algorithm to skip over trace

event	posns
login attempt	[0]
guest login	[1]
auth failed	[2]
authorized	[3]



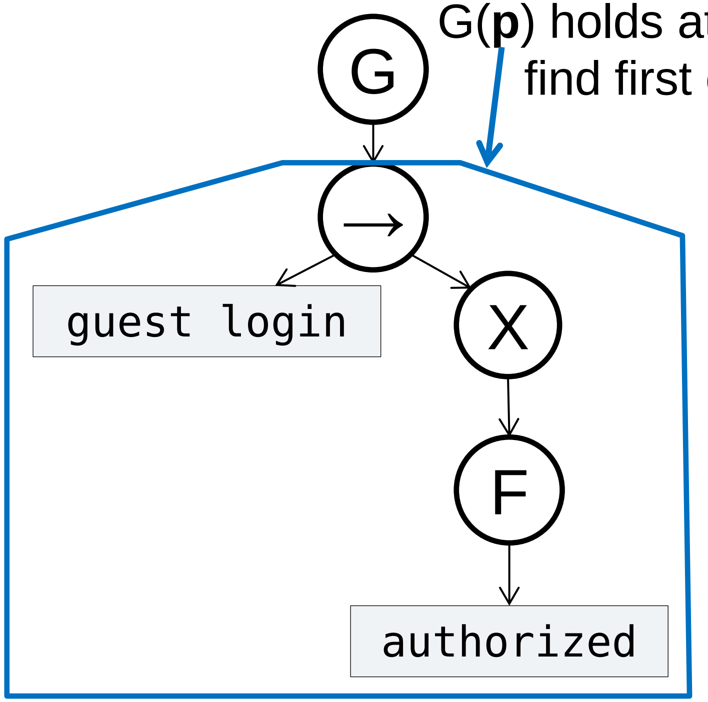
Checking on Map Traces

- Check on trace in map form also tree-based
  - but also uses the negation of nodes
- Map form allows algorithm to skip over trace

event	posns
login attempt	[0]
guest login	[1]
auth failed	[2]
authorized	[3]

G(**p**) holds at 0 if **!p** never occurs

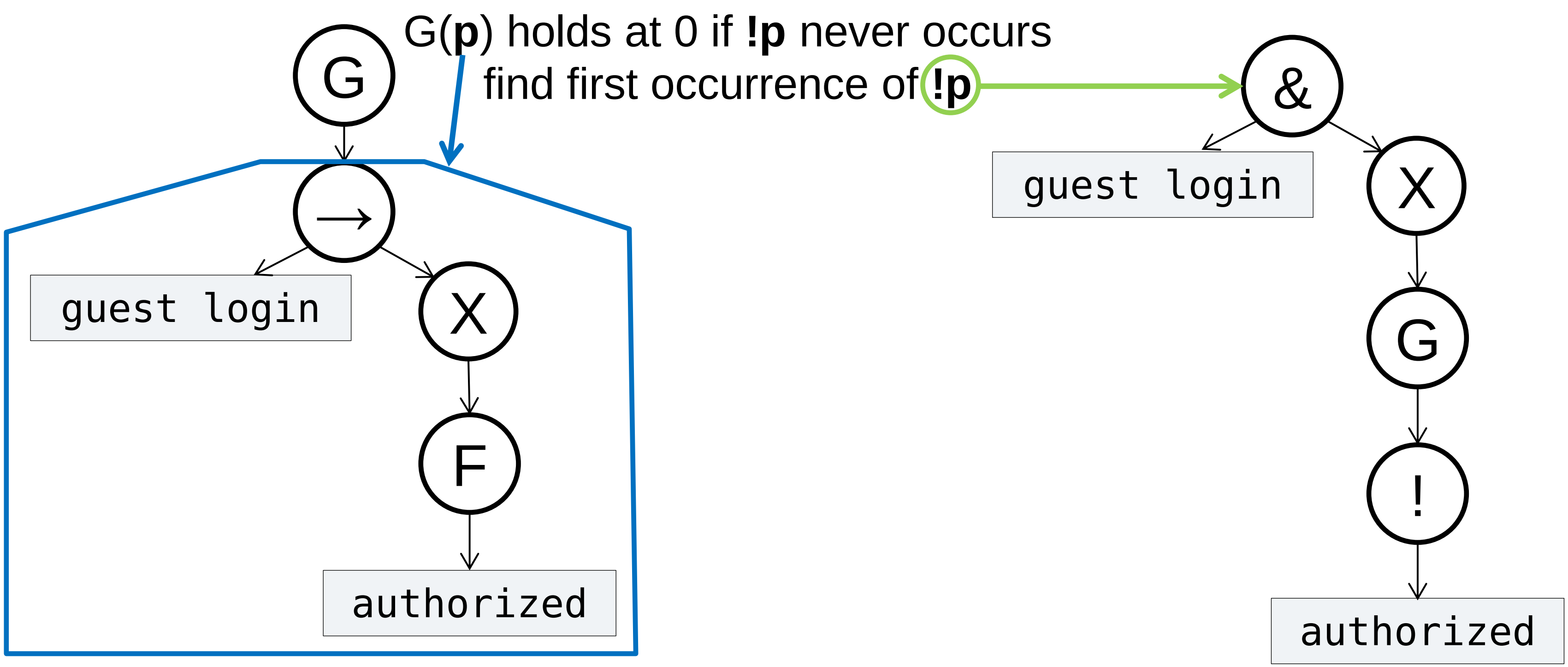
find first occurrence of **!p**



Checking on Map Traces

- Check on trace in map form also tree-based
  - but also uses the negation of nodes
- Map form allows algorithm to skip over trace

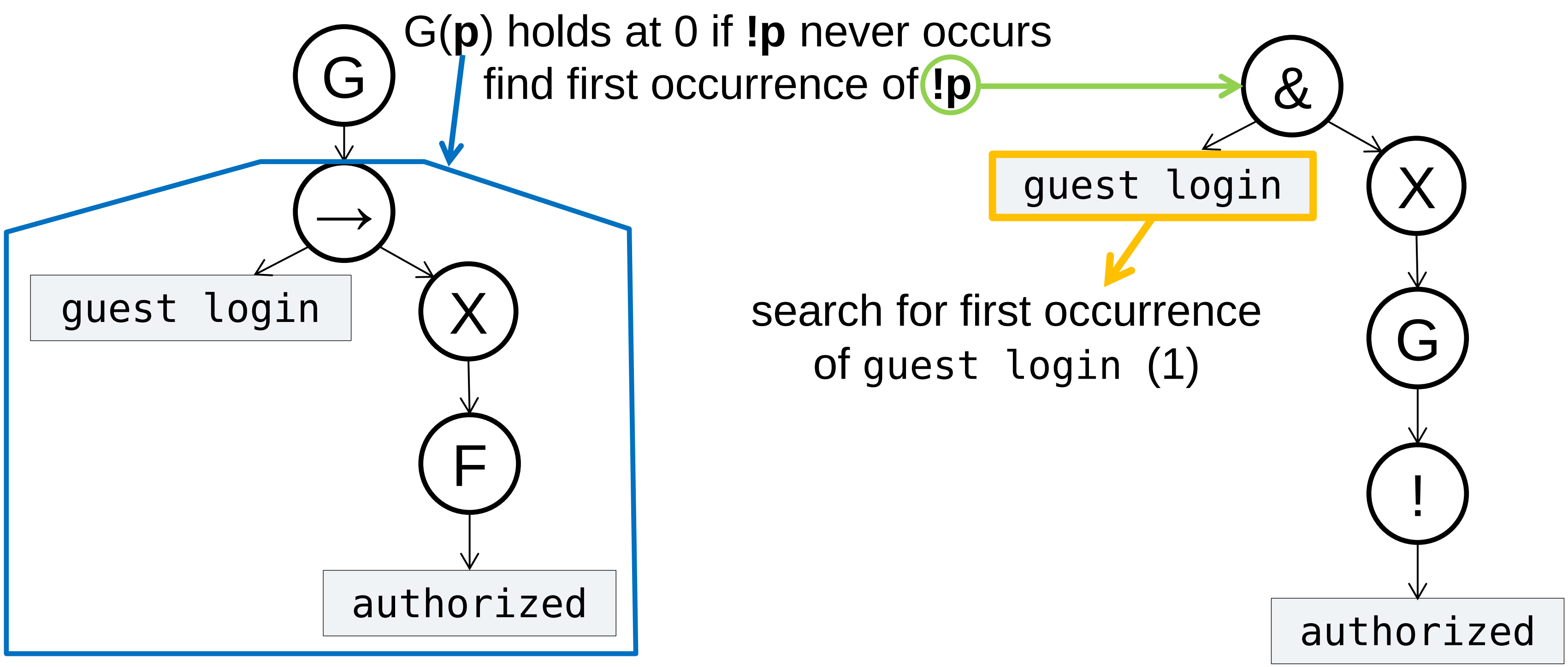
event	posns
login attempt	[0]
guest login	[1]
auth failed	[2]
authorized	[3]



Checking on Map Traces

- Check on trace in map form also tree-based
  - but also uses the negation of nodes
- Map form allows algorithm to skip over trace

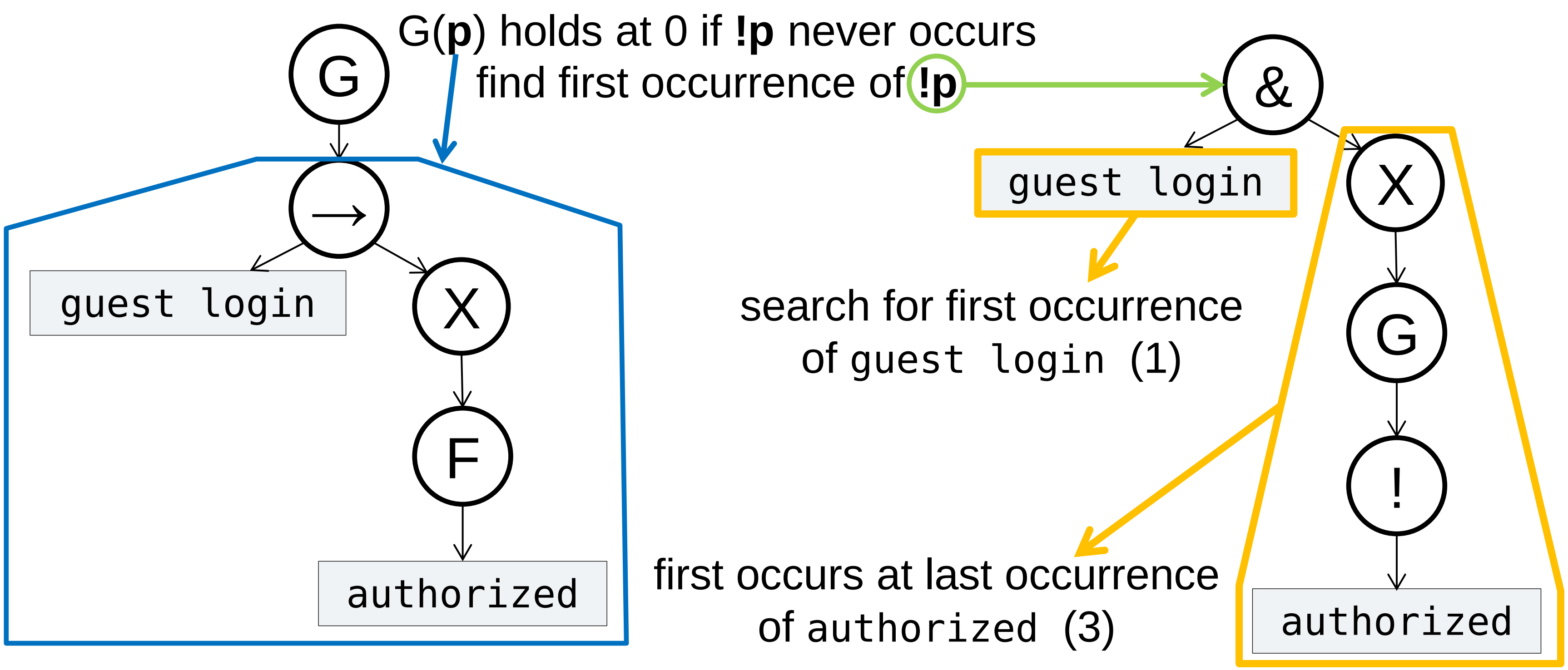
event	**posns**
`login attempt`	`[0]`
`guest login`	`[1]`
`auth failed`	`[2]`
`authorized`	`[3]`



Checking on Map Traces

- Check on trace in map form also tree-based
  - but also uses the negation of nodes
- Map form allows algorithm to skip over trace

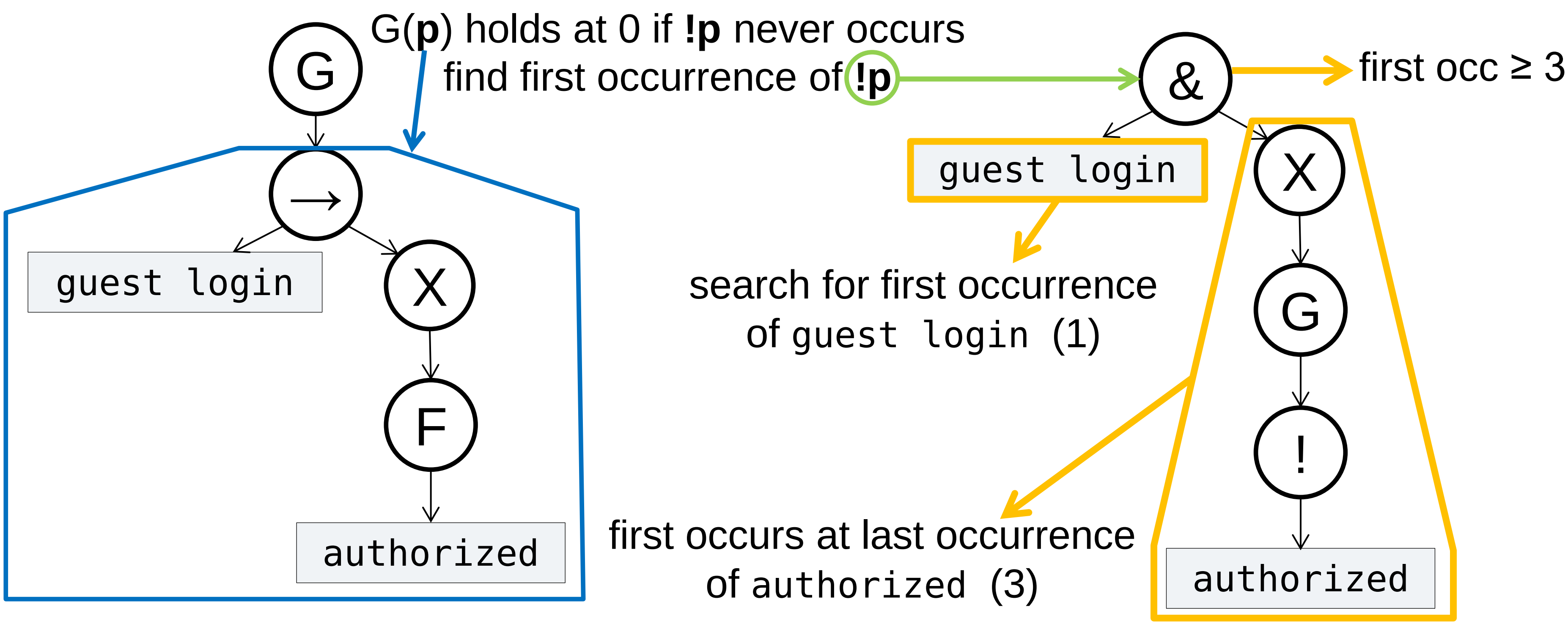
event	posns
`login attempt`	`[0]`
`guest login`	`[1]`
`auth failed`	`[2]`
`authorized`	`[3]`



Checking on Map Traces

- Check on trace in map form also tree-based
  - but also uses the negation of nodes
- Map form allows algorithm to skip over trace

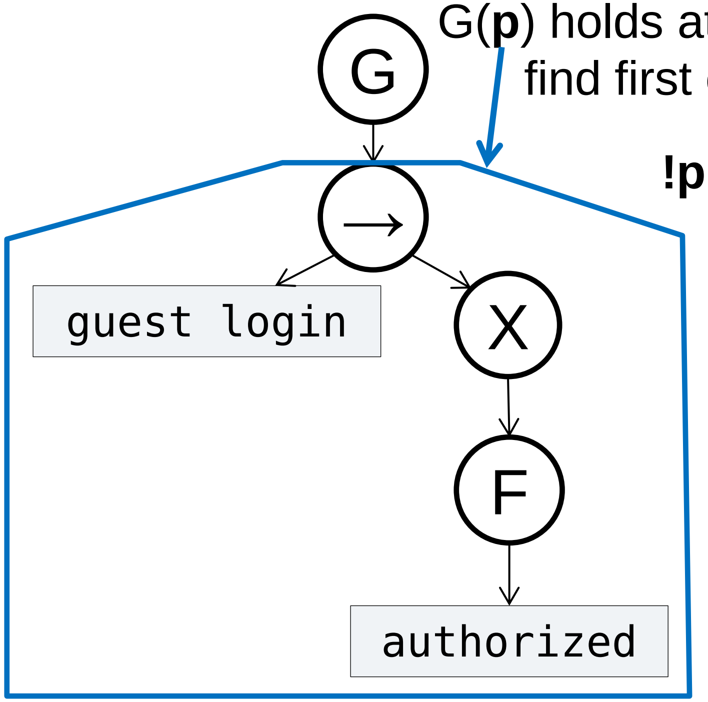
event	posns
`login attempt`	`[0]`
`guest login`	`[1]`
`auth failed`	`[2]`
`authorized`	`[3]`



Checking on Map Traces

- Check on trace in map form also tree-based
  - but also uses the negation of nodes
- Map form allows algorithm to skip over trace

event	posns
login attempt	[0]
guest login	[1]
auth failed	[2]
authorized	[3]



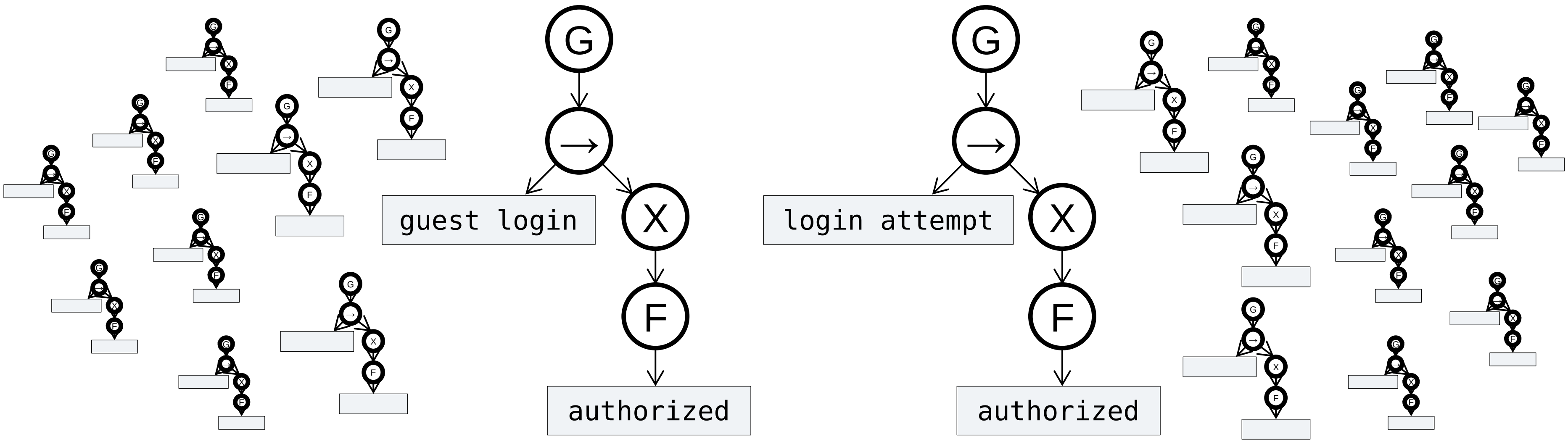
G(**p**) holds at 0 if **!p** never occurs

find first occurrence of **!p**

!p never occurs in trace, G(**p**) holds.

Memoization (reuse of computation)

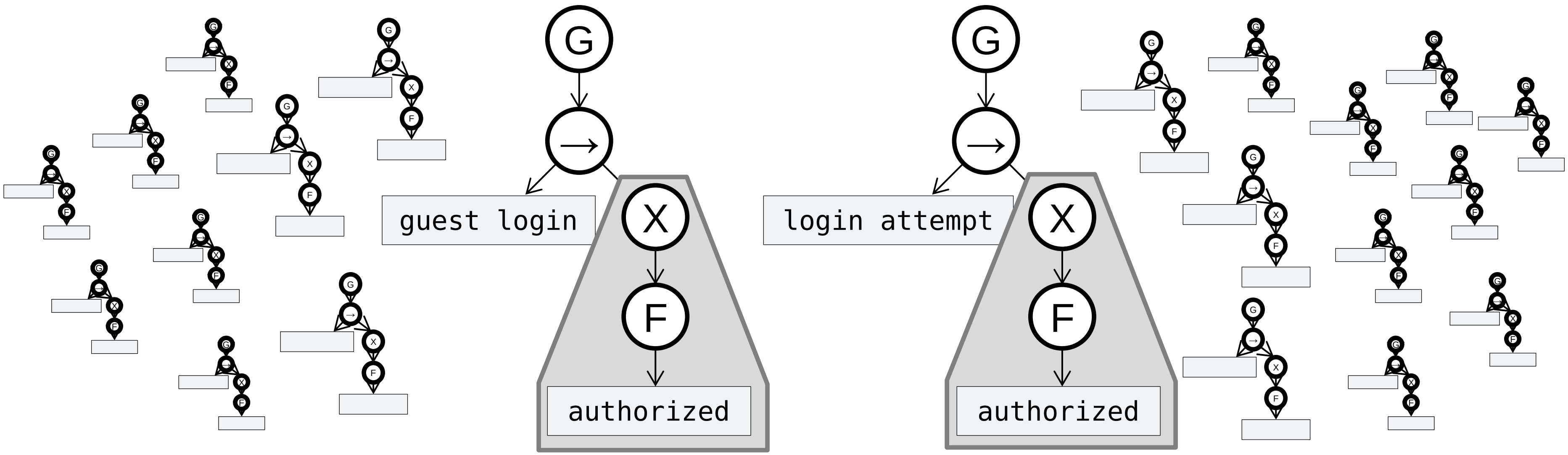
- To check property type, check each instance on log
  - for N unique events, M variables, ~N^M instances
  - tree form allows for specialized memoization



- Preliminary memo over 3 instantiations: 7% speedup

Memoization (reuse of computation)

- To check property type, check each instance on log
  - for N unique events, M variables, ~N^M instances
  - tree form allows for specialized memoization



- Preliminary memo over 3 instantiations: 7% speedup

Support, Confidence for LTL

- Want to know which instances "almost never" violated
- check `guest login` is always followed by `authorized`:

```
login attempt
guest login
auth failed
authorized
guest login
authorized
guest login
```

only one guest login not followed by authorized – guest login is almost always followed by authorized

- Can we formalize this?

Initial Support, Confidence Concept

- Proposal: support for G(**p**) = # number of time points where **p** holds

qqqq	**qpqq**	**pppp**
sup G(**p**)= 0	sup G(**p**)= 1	sup G(**p**)= 4

- But: support for G(**p**→XF**q**)

pppq	**pqpp**	**rrrr**
sup G(**p**→XF**q**)= 4	sup G(**p**→XF**q**)= 2	sup G(**p**→XF**q**)= 4

Initial Support, Confidence Concept

- Proposal: support for G(**p**) = # number of time points where **p** holds

qqqq	**qpqq**	**pppp**
sup G(**p**)= 0	sup G(**p**)= 1	sup G(**p**)= 4

- But: support for G(**p**→XF**q**)

pppq	**pqpp**	**rrrr**
sup G(**p**→XF**q**)= 4	sup G(**p**→XF**q**)= 2	sup G(**p**→XF**q**)= 4

Initial Support, Confidence Concept

- Proposal: support for G(**p**) = # number of time points where **p** holds

qqqq	**qpqq**	**pppp**
sup G(**p**)= 0	$\boxed{\text{sup G}(\mathbf{p})= 1}$	sup G(**p**)= 4

- But: support for G(**p**→XF**q**)

pppq	**pqpp**	**rrrr**
sup G(**p**→XF**q**)= 4	sup G(**p**→XF**q**)= 2	sup G(**p**→XF**q**)= 4

Initial Support, Confidence Concept

- Proposal: support for G(**p**) = # number of time points where **p** holds

qqqq	qpqq	pppp
sup G(**p**)= 0	sup G(**p**)= 1	sup G(**p**)= 4

- But: support for G(**p**→XF**q**)

pppq	pqpp	rrrr
sup G(**p**→XF**q**)= 4	sup G(**p**→XF**q**)= 2	sup G(**p**→XF**q**)= 4

Initial Support, Confidence Concept

- Proposal: support for G(**p**) = # number of time points where **p** holds

qqqq	**qpqq**	**pppp**
sup G(**p**)= 0	sup G(**p**)= 1	sup G(**p**)= 4

- But: support for G(**p**→XF**q**)

pppq	**pqpp**	**rrrr**
sup G(**p**→XF**q**)= 4	sup G(**p**→XF**q**)= 2	sup G(**p**→XF**q**)= 4

Initial Support, Confidence Concept

- Proposal: support for G(**p**) = # number of time points where **p** holds

qqqq	**qpqq**	**pppp**
sup G(**p**)= 0	sup G(**p**)= 1	sup G(**p**)= 4

- But: support for G(**p**→XF**q**)

pppq	**pqpp**	**rrrr**
sup G(**p**→XF**q**)= 4	sup G(**p**→XF**q**)= 2	sup G(**p**→XF**q**)= 4

Initial Support, Confidence Concept

- Proposal: support for G(**p**) = # number of time points where **p** holds

qqqq sup G(**p**)= 0
qpqq sup G(**p**)= 1
pppp sup G(**p**)= 4

- But: support for G(**p**→XF**q**)

pppq sup G(**p**→XF**q**)= 4
pqpp sup G(**p**→XF**q**)= 2
rrrr sup G(**p**→XF**q**)= 4

Support, Confidence Heuristic

- What we do: focus on **falsifiability**

`guest login` $\rightarrow$ XF`authorized`
vacuously true on

```
login attempt
guest login
auth failed
authorized
guest login
authorized
guest login
```

- Call these vacuously true time points ***not falsifiable***
- Approximate support, support potential for arbitrary LTL
  - **Support potential** of Ψ: number of ***falsifiable*** time points
  - **Support** of Ψ: number of ***falsifiable*** time points on which Ψ is satisfied
  - **Confidence** of Ψ: support/support potential (or 1 if both are 0)

Texada Evaluation

- Can Texada mine a wide enough variety of temporal properties?
- Can Texada help comprehend unknown systems?
  - Real estate web log
  - StackAr
- Can Texada confirm expected behavior of systems?
  - Dining Philosophers
  - Sleeping Barber
- Is Texada fast?
  - Texada vs. Synoptic
  - Texada vs. Perracotta
- Can we use Texada's results to build other tools?
  - Quarry prototype

Texada Evaluation

- Can Texada mine a wide enough variety of temporal properties?
- Can Texada help comprehend unknown systems?
  - Real estate web log
  - StackAr
- Can Texada confirm expected behavior of systems?
  - Dining Philosophers
  - Sleeping Barber
- Is Texada fast?
  - Texada vs. Synoptic
  - Texada vs. Perracotta NEW
- Can we use Texada's results to build other tools?
  - Quarry prototype

Expressiveness of Property Types

- Texada can express properties from prior work

	Name	Regex	LTL
– Synoptic[1]	Always Followed by		G(x→XFy)
	Never Followed by		G(x→XG!y)
	Always Precedes		(!y W x)
– Perracotta[2]	Alternating	(xy)*	(!y W x) & G((x→X(!x U y)) & (y→ X(!y W x)))
	MultiEffect	(xyy*)*	(!y W x) & G(x→X(!x U y))
	MultiCause	(xx*y)*	(!y W x) & G((x→XFy) & (y→X(!y W x)))
	EffectFirst	y*(xy)*	G((x→X(!x U y)) & (y→ X(!y W x)))
	OneCause	y*(xyy*)*	G(x→X(!x U y))
	CauseFirst	(xx*yy*)*	(!y W x) & G(x→XFy)
	OneEffect	y*(xx*y)*	G((x→XFy) & (y→X(!y W x)))

  - *Patterns in Property Specifications for Finite-State Verification* [Dwyer et al. ICSE'99]

[1] I. Beschastnikh, Y. Brun, S. Schneider, M. Sloan and M. D. Ernst. Leveraging Existing Instrumentation to Automatically Infer Invariant-Constrained Models. FSE11.

[2] Jinlin Yang, David Evans, Deepali Bhardwaj, Thirumalesh Bhat, Manuvir Das. Perracotta: Mining Temporal API Rules from Imperfect Traces. ICSE06.

Expressiveness of Property Types

- Texada can express properties from prior work

Name	Regex	LTL
Always Followed by		G(x→XFy)
CauseFirst	(xx*yy*)*	(!y W x) & G(x→XFy)
OneEffect	y*(xx*y)*	G((x→XFy) & (y→X(!y W x)))

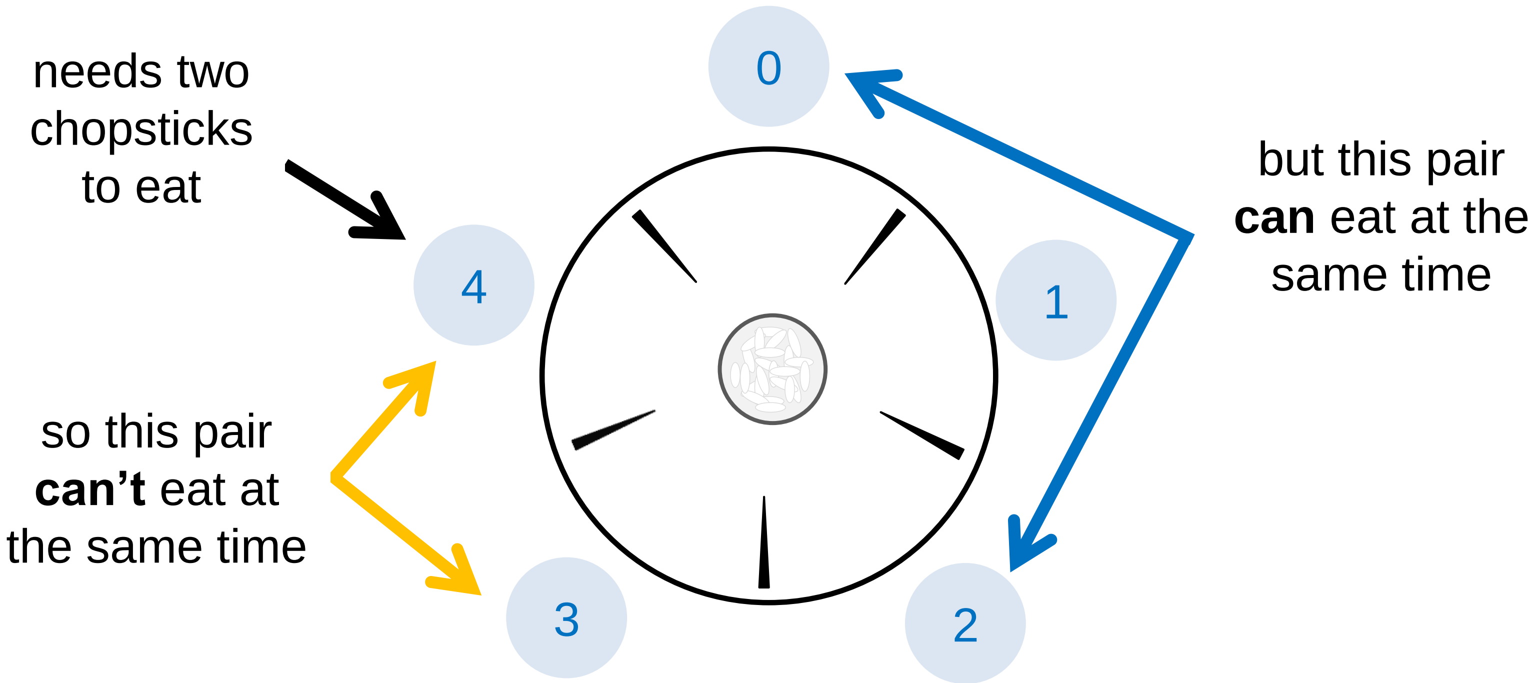
- *Patterns in Property Specifications for Finite-State Verification* [Dwyer et al. ICSE'99]

- Texada can mine a wide variety of properties ✓
- Texada can mine concurrent sys. properties
- Texada has reasonable performance

[1] I. Beschastnikh, Y. Brun, S. Schneider, M. Sloan and M. D. Ernst. Leveraging Existing Instrumentation to Automatically Infer Invariant-Constrained Models. FSE11.
[2] Jinlin Yang, David Evans, Deepali Bhardwaj, Thirumalesh Bhat, Manuvir Das. Perracotta: Mining Temporal API Rules from Imperfect Traces. ICSE06.

Dining Philosophers

- Classic concurrency problem: philosophers sit around a table, thinking, hungry, or eating.



- These specs could not be checked with previous temporal spec miners!

Multi-Propositional Traces

- LTL: multiple atomic propositions may hold at a time
- Standard log model: **one event at each time point**
- Texada supports multi-propositional logs: **multiple events can occur at one time point**
- Dining philosophers log: 5 one minute traces, 6.5K lines

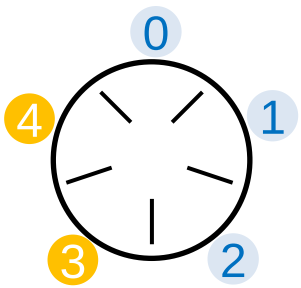
multiple events at single time point

time point separator

```
0 is THINKING
1 is HUNGRY
2 is THINKING
3 is THINKING
4 is THINKING
..
0 is THINKING
1 is EATING
2 is THINKING
3 is THINKING
4 is THINKING
..
      ...
```

Dining Phil. Mutex (safety property)

- Two adjacent philosophers never eat at the same time
- Property pattern: G($x \rightarrow !y$) "if x occurs, y does not"



G(3 is EATING →! 4 is EATING)

⬍

G(4 is EATING →! 3 is EATING)

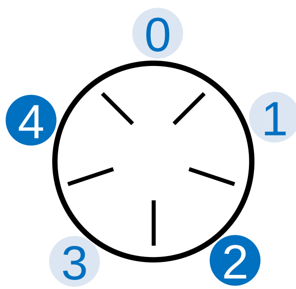
- Texada output for G($x \rightarrow !y$) includes

G(0 is EATING →! 1 is EATING)
G(0 is EATING →! 4 is EATING)
G(1 is EATING →! 2 is EATING)
G(2 is EATING →! 3 is EATING)
G(3 is EATING →! 4 is EATING)

together, mean that two adjacent philosophers never eat at the same time

Dining Phil. Efficiency (liveness property)

- Non-adjacent philosophers eventually eat at the same time
- Property pattern: F(*x* & *y*) "eventually *x* and *y* occur together"




```
F(2 is EATING & 4 is EATING)
```

⬍

```
F(4 is EATING & 2 is EATING)
```

- Texada output for F(*x* & *y*) includes

```
F(0 is EATING & 2 is EATING)
F(0 is EATING & 3 is EATING)
F(1 is EATING & 3 is EATING)
F(1 is EATING & 4 is EATING)
F(2 is EATING & 4 is EATING)
```

together, mean that non-adjacent philosophers eventually eat at the same time

Dining Phil. Efficiency (liveness property)

- Non-adjacent philosophers eventually eat at the same time
- Property pattern: F(*x* & *y*) “eventually *x* and *y* occur together”

- Texada can mine a wide variety of properties
- Texada can mine concurrent sys. properties
- Texada has reasonable performance

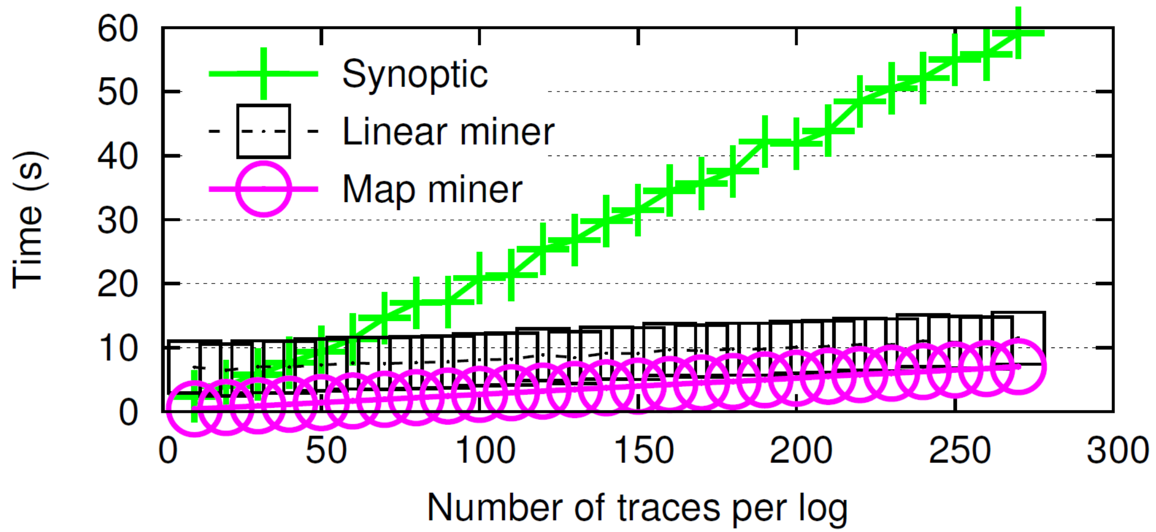
- T

```
F(0 is EATING & 2 is EATING)
F(0 is EATING & 3 is EATING)
F(1 is EATING & 3 is EATING)
F(1 is EATING & 4 is EATING)
F(2 is EATING & 4 is EATING)
```

together, mean that non-adjacent philosophers eventually eat at the same time

Texada vs. Synoptic

- Texada performs favourably against Synoptic's miner on three property types it is *specialized* to mine.



- More results in paper.
- Texada algs benefit from log-level short-circuiting.

Texada vs. Perracotta

- Perracotta performs favourably against Texada:

Unique events (10K events/trace, 20 traces/log)	**Perracotta**	**Texada (map miner)**
120	0.85 s	2.42 s
160	0.97 s	4.07 s
260	1.42 s	10.21 s

- Perracotta's algorithm particularly effective at reducing instantiation effect on runtime.
- Further memoization work (along with good expiration policies) might help reduce instantiation effect

Texada vs. Perracotta

- Perracotta performs favourably against Texada:

Unique events (10K events/trace, 20	Perracotta	Texada (map miner)

- Texada can mine a wide variety of properties ✓
- Texada can mine concurrent sys. properties ✓
- Texada has reasonable performance ✓

- Perracotta's algorithm particularly effective at reducing instantiation effect on runtime.
- Further memoization work (along with good expiration policies) might help reduce instantiation effect

Conclusion

- Many temporal spec miners, unclear which to use
- Texada: general LTL spec miner
  - confirms expected behavior, discovers unexpected use patterns
  - prototyped confidence measures (future work to improve this)
  - can examine concurrent system logs
- Open source and ready to use:

https://bitbucket.org/bestchai/texada/